Synchronization: Advanced

15-213 / 18-213: Introduction to Computer Systems
25th Lecture, Nov. 21, 2017

Instructor:
  Randy Bryant
Reminder: Semaphores

- **Semaphore**: non-negative global integer synchronization variable

- Manipulated by \( P \) and \( V \) operations:
  - \( P(s) \): \begin{align*} &\text{[ while } (s == 0) \text{ wait(); } s--; \text{ ]} \\ &\text{Dutch for "Proberen" (test)} \end{align*}
  - \( V(s) \): \begin{align*} &\text{[ } s++; \text{ ]} \\ &\text{Dutch for "Verhogen" (increment)} \end{align*}

- OS kernel guarantees that operations between brackets \([ \] \) are executed atomically
  - Only one \( P \) or \( V \) operation at a time can modify \( s \).
  - When \textbf{while} loop in \( P \) terminates, only that \( P \) can decrement \( s \)

- **Semaphore invariant**: \( (s >= 0) \)
Review: Using semaphores to protect shared resources via mutual exclusion

Basic idea:

- Associate a unique semaphore `mutex`, initially 1, with each shared variable (or related set of shared variables)
- Surround each access to the shared variable(s) with `P(mutex)` and `V(mutex)` operations

```
mutex = 1
P(mutex)
cnt++
V(mutex)
```

- This case is so common, that pthreads provides mutex as primitive.
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem
  - Readers-writers problem

- Other concurrency issues
  - Thread safety
  - Races
  - Deadlocks
Using Semaphores to Coordinate Access to Shared Resources

- **Basic idea:** Thread uses a semaphore operation to notify another thread that some condition has become true
  - Use counting semaphores to keep track of resource state.
  - Use binary semaphores to notify other threads.

- **Two classic examples:**
  - The Producer-Consumer Problem
  - The Readers-Writers Problem
**Producer-Consumer Problem**

- **Common synchronization pattern:**
  - Producer waits for empty *slot*, inserts item in buffer, and notifies consumer
  - Consumer waits for *item*, removes it from buffer, and notifies producer

- **Examples**
  - Multimedia processing:
    - Producer creates video frames, consumer renders them
  - Event-driven graphical user interfaces
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - Consumer retrieves events from buffer and paints the display
Producer-Consumer on 1-element Buffer

- Maintain two semaphores: `full` + `empty`
Producer-Consumer on 1-element Buffer

```c
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;

int main(int argc, char** argv) {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* Initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);

    /* Create threads and wait */
    Pthread_create(&tid_producer, NULL, producer, NULL);
    Pthread_create(&tid_consumer, NULL, consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    return 0;
}
```
Producer-Consumer on 1-element Buffer

Initially: empty==1, full==0

**Producer Thread**

```c
void *producer(void *arg) {
    int i, item;
    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n", item);
        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

**Consumer Thread**

```c
void *consumer(void *arg) {
    int i, item;
    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);
        /* Consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```
Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers

- Producers will contend with each to get \texttt{empty}
- Consumers will contend with each other to get \texttt{full}

Producers
\begin{verbatim}
P(&\texttt{shared.empty});
shared.buf = item;
V(&\texttt{shared.full});
\end{verbatim}

Consumers
\begin{verbatim}
P(&\texttt{shared.full});
item = shared.buf;
V(&\texttt{shared.empty});
\end{verbatim}
Producer-Consumer on an $n$-element Buffer

- Implemented using a shared buffer package called sbuf.
Circular Buffer (n = 10)

- Store elements in array of size n
- items: number of elements in buffer
- Empty buffer:
  - front = rear
- Nonempty buffer
  - rear: index of most recently inserted element
  - front: (index of next element to remove – 1) mod n
- Initially:

<table>
<thead>
<tr>
<th>front</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rear</td>
<td>0</td>
</tr>
<tr>
<td>items</td>
<td>0</td>
</tr>
</tbody>
</table>

```
0 1 2 3 4 5 6 7 8 9
```
Circular Buffer Operation (n = 10)

- **Insert 7 elements**
  - front: 0
  - rear: 7
  - items: 7

- **Remove 5 elements**
  - front: 5
  - rear: 7
  - items: 2

- **Insert 6 elements**
  - front: 5
  - rear: 3
  - items: 8

- **Remove 8 elements**
  - front: 3
  - rear: 3
  - items: 0
Sequential Circular Buffer Code

```
init(int v)
{
    items = front = rear = 0;
}

insert(int v)
{
    if (items >= n)
        error();
    if (++rear >= n) rear = 0;
    buf[rear] = v;
    items++;
}

int remove()
{
    if (items == 0)
        error();
    if (++front >= n) front = 0;
    int v = buf[front];
    items--;
    return v;
}
```
Producer-Consumer on an $n$-element Buffer

- **Requires a mutex and two counting semaphores:**
  - **mutex**: enforces mutually exclusive access to the buffer and counters
  - **slots**: counts the available slots in the buffer
  - **items**: counts the available items in the buffer

- **Makes use of general semaphores**
  - Will range in value from 0 to $n$
#include "csapp.h"

typedef struct {
    int *buf;       /* Buffer array */
    int n;          /* Maximum number of slots */
    int front;      /* buf[front+1 (mod n)] is first item */
    int rear;       /* buf[rear] is last item */
    sem_t mutex;    /* Protects accesses to buf */
    sem_t slots;    /* Counts available slots */
    sem_t items;    /* Counts available items */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
sbuf Package - Implementation

Initializing and deinitializing a shared buffer:

```c
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n; /* Buffer holds max of n items */
    sp->front = sp->rear = 0; /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

sbuf Package - Implementation

Inserting an item into a shared buffer:

```c
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);               /* Wait for available slot */
    P(&sp->mutex);               /* Lock the buffer */
    if (++sp->rear >= sp->n)     /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item;    /* Insert the item */
    V(&sp->mutex);               /* Unlock the buffer */
    V(&sp->items);               /* Announce available item */
}
```

sbuf.c
Removing an item from a shared buffer:

```c
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                         /* Wait for available item */
    P(&sp->mutex);                         /* Lock the buffer */
    if (++sp->front >= sp->n)             /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];           /* Remove the item */
    V(&sp->mutex);                        /* Unlock the buffer */
    V(&sp->slots);                        /* Announce available slot */
    return item;
}
```

sbuf.c
Demonstration

- See program produce-consume.c in code directory
- 10-entry shared circular buffer
- 5 producers
  - Agent i generates numbers from 20*i to 20*i – 1.
  - Puts them in buffer
- 5 consumers
  - Each retrieves 20 elements from buffer
- Main program
  - Makes sure each value between 0 and 99 retrieved once
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem
  - Readers-writers problem

- Other concurrency issues
  - Thread safety
  - Races
  - Deadlocks
Readers-Writers Problem

Problem statement:
- **Reader** threads only read the object
- **Writer** threads modify the object (read/write access)
- Writers must have exclusive access to the object
- Unlimited number of readers can access the object

Occurs frequently in real systems, e.g.,
- Online airline reservation system
- Multithreaded caching Web proxy
Readers/Writers Examples
Variants of Readers-Writers

- **First readers-writers problem** *(favors readers)*
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.

- **Second readers-writers problem** *(favors writers)*
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.

- **Starvation** *(where a thread waits indefinitely)* is possible in both cases.
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt; /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

rw1.c
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w;    /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w;    /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

- Readcnt == 2
- W == 0
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt; /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0
# Solution to First Readers-Writers Problem

**Readers:**

```c
int readcnt; /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
    
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

**Writers:**

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

**Arrivals:** R1 R2 W1 R3

Readcnt == 1
W == 0
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;  /* Initially 0 */
sem_t mutex, w;  /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt; /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        /* Reading happens here */
        V(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
        /* Reading happens here */
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 0
W == 1
Other Versions of Readers-Writers

- **Shortcoming of first solution**
  - Continuous stream of readers will block writers indefinitely

- **Second version**
  - Once writer comes along, blocks access to later readers
  - Series of writes could block all reads

- **FIFO implementation**
  - See rwqueue code in code directory
  - Service requests in order received
  - Threads kept in FIFO
  - Each has semaphore that enables its access to critical section
Solution to Second Readers-Writers Problem

```c
int readcnt, writecnt; // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
    while (1) {
        P(&r);
        P(&rmutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&rmutex);
        V(&r)
        /* Reading happens here */
        P(&rmutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&rmutex);
    }
}
```
Solution to Second Readers-Writers Problem

```c
void writer(void) {
    while (1) {
        P(&wmutex);
        writecnt++;
        if (writecnt == 1)
            P(&r);
        V(&wmutex);

        P(&w);
        /* Writing here */
        V(&w);

        P(&wmutex);
        writecnt--;
        if (writecnt == 0)
            V(&r);
        V(&wmutex);
    }
}
```
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem
  - Readers-writers problem

- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
One Worry: Races

- A **race** occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y.

```c
/* a threaded program with a race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

race.c
Data Race
Race Elimination

Make sure don’t have unintended sharing of state

```c
/* a threaded program without the race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = malloc(sizeof(int));
        *valp = i;
        pthread_create(&tid[i], NULL, thread, valp);
    }
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

norace.c
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem
  - Readers-writers problem

- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
A Worry: Deadlock

- Def: A process is *deadlocked* iff it is waiting for a condition that will never be true.

- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!
Deadlocking With Semaphores

```c
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    return 0;
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
- P(s₀);
- P(s₁);
- cnt++;
- V(s₀);
- V(s₁);

Tid[1]:
- P(s₁);
- P(s₀);
- cnt++;
- V(s₁);
- V(s₀);
Deadlock Visualized in Progress Graph

Locking introduces the potential for deadlock: waiting for a condition that will never be true.

Any trajectory that enters the deadlock region will eventually reach the deadlock state, waiting for either $s_0$ or $s_1$ to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often nondeterministic (race).
Deadlock
Avoiding Deadlock  

Acquire shared resources in same order

```c
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    return 0;
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s₀);
P(s₁);
cnt++;
V(s₀);
V(s₁);

Tid[1]:
P(s₀);
P(s₁);
cnt++;
V(s₁);
V(s₀);
Avoided Deadlock in Progress Graph

No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial
Demonstration

- See program deadlock.c
- 100 threads, each acquiring same two locks
- Risky mode
  - Even numbered threads request locks in opposite order of odd-numbered ones

- Safe mode
  - All threads acquire locks in same order
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem
  - Readers-writers problem

- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
Crucial concept: Thread Safety

- Functions called from a thread must be *thread-safe*

- **Def:** A function is *thread-safe* iff it will always produce correct results when called repeatedly from multiple concurrent threads.

- **Classes of thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions
Thread-Unsafe Functions (Class 1)

- Failing to protect shared variables
  - Fix: Use $P$ and $V$ semaphore operations
  - Example: `goodcnt.c`
  - Issue: Synchronization operations will slow down code
Thread-Unsafe Functions (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator that relies on static state

```c
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```
Thread-Safe Random Number Generator

- Pass state as part of argument
  - and, thereby, eliminate static state

```c
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- Consequence: programmer using `rand_r` must maintain seed
Thread-Unsafe Functions (Class 3)

- Returning a pointer to a static variable
  - Requires changes in caller and callee
- Fix 1. Rewrite function so caller passes address of variable to store result
  - Requires changes in caller and callee
- Fix 2. Lock-and-copy
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory.

```c
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    sprintf(buf, "%d", x);
    return buf;
}
```

```c
char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```

Warning: Some functions like gethostbyname require a deep copy. Use reentrant gethostbyname_r version instead.
Thread-Unsafe Functions (Class 4)

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions 😊
Reentrant Functions

- **Def:** A function is *reentrant* iff it accesses no shared variables when called by multiple threads.
  - Important subset of thread-safe functions
    - Require no synchronization operations
    - Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

### All functions

<table>
<thead>
<tr>
<th>Thread-safe functions</th>
<th>Thread-unsafe functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrant functions</td>
<td></td>
</tr>
</tbody>
</table>
Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - Examples: `malloc, free, printf, scanf`

- Most Unix system calls are thread-safe, with a few exceptions:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>strftime</td>
<td>3</td>
<td>strftime_r</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_pton</td>
<td></td>
<td>(none)</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threads Summary

- Threads provide another mechanism for writing concurrent programs
- Threads are growing in popularity
  - Somewhat cheaper than processes
  - Easy to share data between threads
- However, the ease of sharing has a cost:
  - Easy to introduce subtle synchronization errors
  - Tread carefully with threads!

- For more info:
  - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997