

---

# CS 241 System Programming

## More Timers, Signals

### Midterm review

---

Discussion Section 6  
27 Feb – 2 Mar

---

# Outline

- Signals & Timers revisited
  - Midterm Review
    - Processes
    - Threads
    - Synchronization
    - Scheduling
-

---

# Timers: sigevent

- POSIX:TMR timers by default send SIGALRM
  - Use struct sigevent to change this if needed

```
struct sigevent {  
    int          sigev_notify; /* notification type */  
    int          sigev_signo;  /* signal number */  
    union sigval sigev_value; /* signal value */  
}
```

- sigev\_notify
    - SIGEV\_NONE - No notification from timer
    - SIGEV\_SIGNAL - Default: Send a signal
    - SIGEV\_THREAD - Create a new thread and run it
  - union sigval
    - Either int sigval\_int or void\* sigval\_ptr
-

---

# Alarm signal – sending SIGUSR1

```
timer_t timerid;

void createTimer()
{
    struct sigevent se;
    se.sigev_signo = SIGUSR1;
    if (timer_create(CLOCK_REALTIME, &se, &timerid)==-1)
    {
        fprintf(stderr, "Failed to create timer\n");
        exit(-1);
    }
}
```

---

---

# Starting/stopping timer

```
void setTimer(int seconds)
{
    struct itimerspec timervals;

    timervals.it_value.tv_sec = seconds;
    timervals.it_value.tv_nsec = 0;
    timervals.it_interval.tv_sec = seconds;
    timervals.it_interval.tv_nsec = 0;

    if (timer_settime(timerid, 0, &timervals, NULL) == -1)
    {
        fprintf(stderr, "Failed to start timer\n");
        exit(-1);
    }
}
```

---

---

# sa\_sigaction

(Section 9.4)

```
char* code = NULL;
switch (info->si_code)
{
case SI_USER:      code = "USER" ;      break;
case SI_QUEUE:    code = "QUEUE" ;     break;
case SI_TIMER:    code = "TIMER" ;     break;
case SI_ASYNCIO:  code = "ASYNCIO" ;   break;
case SI_MESGQ:    code = "MESGQ" ;     break;
default:          code = "Unknown" ;   break;
}
```

---

---

# Signal handler for multiple signals

```
void signalCatcher(int signo, siginfo_t* info, void* context)
{
    char message[30];
    char* code = NULL;
    switch (info->si_code) { <..SNIP..from previous slide..> }

    fprintf(stderr, "Signal=%3d(%s), si_signo=%3d, si_code=%d(%s),
        si_value=%d\n", signo, getSignal(signo),
        info->si_signo, info->si_code, code, info->si_value.sival_int);

    /* Actual action depends on signal */
    switch (signo)
    {
    case SIGQUIT /*Ctrl-|*/:      setTimer(0);      break; /* Stop timer */
    case SIGINT /*Ctrl-C*/:      setTimer(1);      break; /* Start timer */
    default: break;
    }
}
```

---

---

# Timers & threads: SIGEV\_THREAD

- At each timer event, create a thread and run `threadfunc0`

```
void createTimer() {
    struct Value* value = (struct Value*)malloc(sizeof(struct Value));
    value->a = 100;
    value->b = 100.001;

    struct sigevent se;
    se.sigev_signo = SIGUSR1;
    se.sigev_notify = SIGEV_THREAD;
    se.sigev_notify_function = threadfunc0;
    se.sigev_value.sival_ptr = (void*)value;
    if (timer_create(CLOCK_REALTIME, &se, &timerid)== -1)
    {
        fprintf(stderr, "Failed to create timer");
        exit(-1);
    }
    fprintf(stderr, "Timer Created\n");
}
```

---

---

# Timer thread example

```
void threadfunc0(union sigval sv)
{
    struct Value* value = (struct Value*)sv.sival_ptr;

    int mycounter = counter++;
    fprintf(stderr, "Running thread: %ud, counter = %d, value->a =
%d, value->b = %f\n",
        pthread_self(), mycounter, value->a, value->b);

    int i;
    for (i = 0; i < 10; i++)
    {
        sleep (1);
        fprintf(stderr, "Running thread: %ud,
            counter = %d\n", pthread_self(), mycounter);
    }
}
```

---

---

# Passing values to signal handlers

- `kill` – Sends a signal to a pid
- `sigqueue` – Queues & sends a signal with a value to a pid
  - `union sigval` value can be `int` or `void*`

```
int main(int argc, char* argv[])
{
    int pid = atoi(argv[1]);
    int signal = getSignal(argv[2]);
    int sendValue = atoi(argv[3]);

    kill(pid, signal); /* Send signal to pid */

    union sigval value;
    value.sival_int = sendValue;
    /* Queues & sends signal with value */
    sigqueue(pid, signal, value);
}
```

---

---

# Outline

- Signals & Timers revisited
  - Midterm Review
    - Processes
    - Threads
    - Synchronization
    - Scheduling
-

# Processes

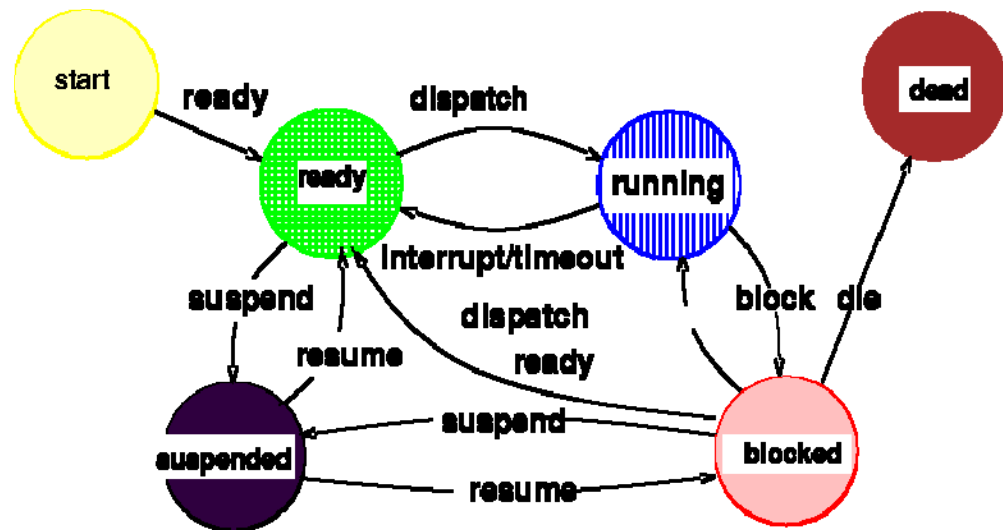
## ■ Process

- ❑ Instance of a program (currently executing)
- ❑ Has alterable state, variables, memory
- ❑ Process ID

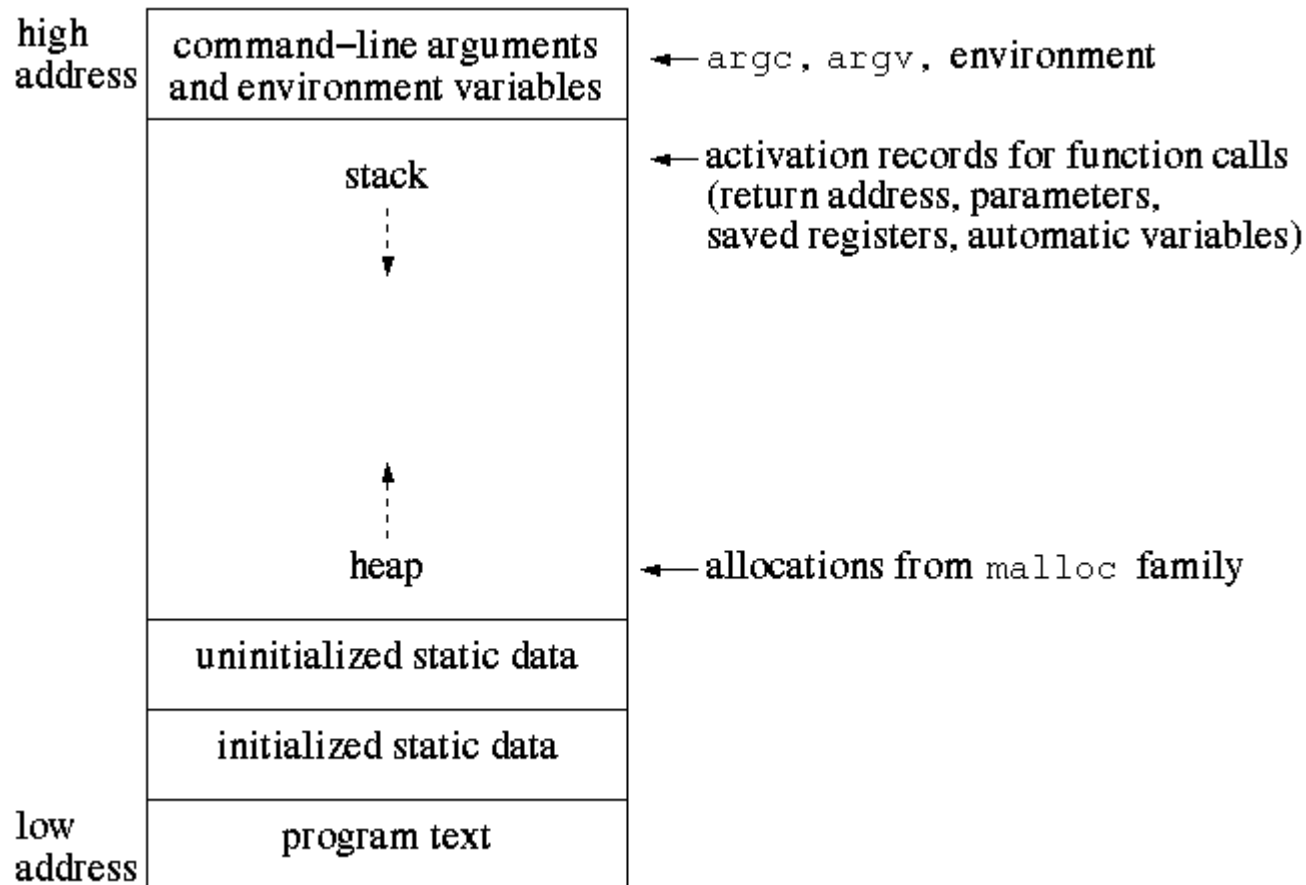
## ❑ States

- New
- Running
- Blocked
- Ready
- Done

## ❑ Context Switches



# Memory Layout



---

# Forking processes (Chains & Fans)

- Process Chain

```
for (i = 1; i < n; i++)  
    if (childpid = fork())  
        break;
```

- Process Fan

```
for (i = 1; i < n; i++)  
    if (((childpid = fork()) <= 0))  
        break;
```

---

---

# Thread

- ADT representing thread of execution within a process
  - Avoids context switches
  - Has its own execution stack, program counter value, register set, and state
  - Share process address space & process resources
  - “Lightweight process”
-

---

# pThreads

```
void* threadfunction(void* arg);
```

<b>POSIX function</b>	<b>Description</b>
pthread_cancel	terminate another thread
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

---

---

# Synchronization - review

- Atomic operations
  - Critical sections
  - Mutual exclusion
  
  - Mutex
  - Semaphore
  - Conditional Variable
  - Read-write locks
-

---

# CPU Scheduling

## ■ Algorithms

- ❑ First-Come First-Serve (FCFS)
- ❑ Shortest Job First (SJF)
- ❑ Round Robin (RR)
- ❑ Priority
- ❑ Preemptive & Non-preemptive

## ■ Metrics

- ❑ Waiting Time
  - ❑ Turnaround Time
-

---

# Queuing Theory

- $\lambda$  = arrival rate
  - $\mu$  = service rate
  - $W_q$  = mean time a customer spends in the queue
  - $W$  = mean time a customer spends in the system
  - $L_q$  = number of customers in queue
  - $L$  = number of customers in the system
  
  - $L_q = \lambda W_q$
  - $L = \lambda W$  ( Little's theorem)
-

# Analysis of Single Server Queue

- Server Utilization:  $\rho = \frac{\lambda}{\mu}$
- Time in System:  $W = \frac{1}{\mu - \lambda}$
- Time in Queue:  $W_q = \frac{\rho}{\mu - \lambda}$
- Number in Queue (Little):  $L_q = \frac{\rho^2}{1 - \rho}$

---

# Queuing Example

Arrival 3 job/sec from Start

Arrival 2 jobs/sec from Event queue

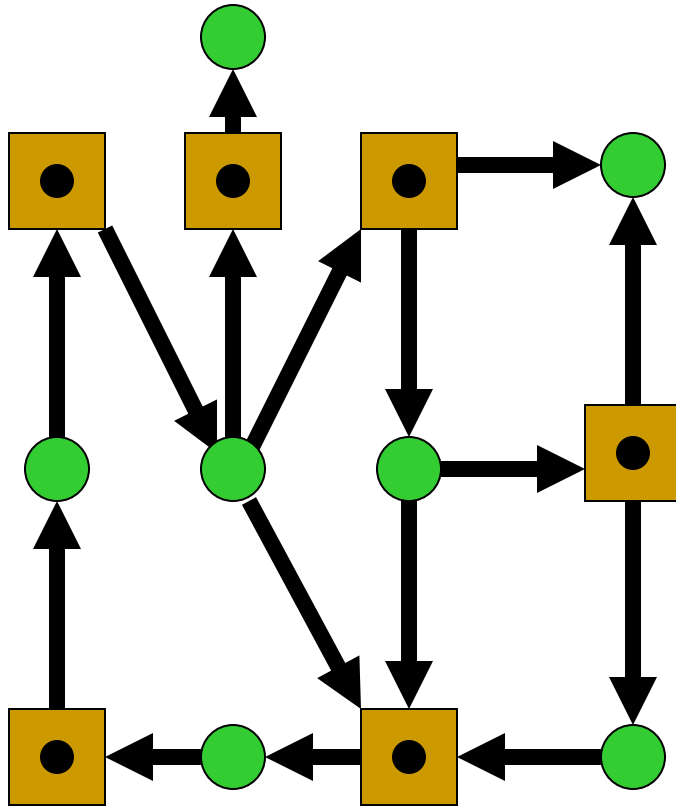
Processor 1 services 2 jobs/sec

Processor 2 services 4 jobs/sec

- Compute the following:
    - Utilization
    - Time in system
    - Time in queue
    - Length of queue
  - Now, what if processor 2 services 3 jobs/sec?
-

---

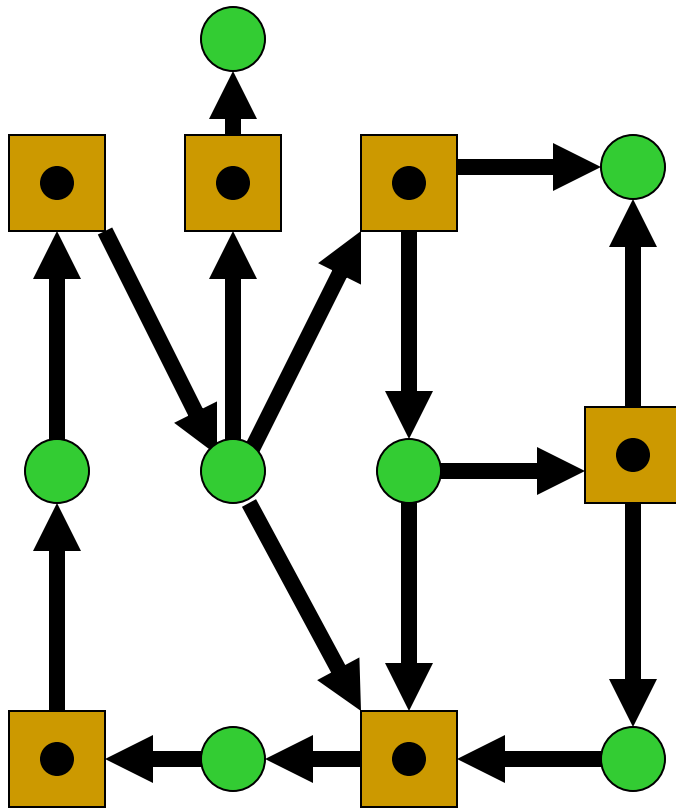
# Resource allocation & Wait for Graphs



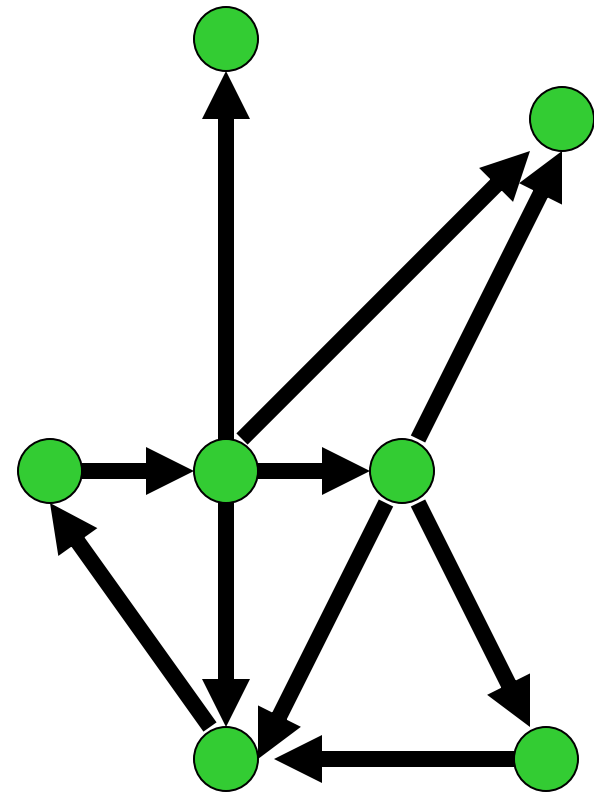
Resource Allocation Graph

---

# Resource allocation & Wait for Graphs



Resource Allocation Graph



Corresponding Wait For Graph

---

# Recap

- Signals & Timers
  - Midterm review
    - Processes
    - Threads
    - Synchronization
    - Scheduling
-