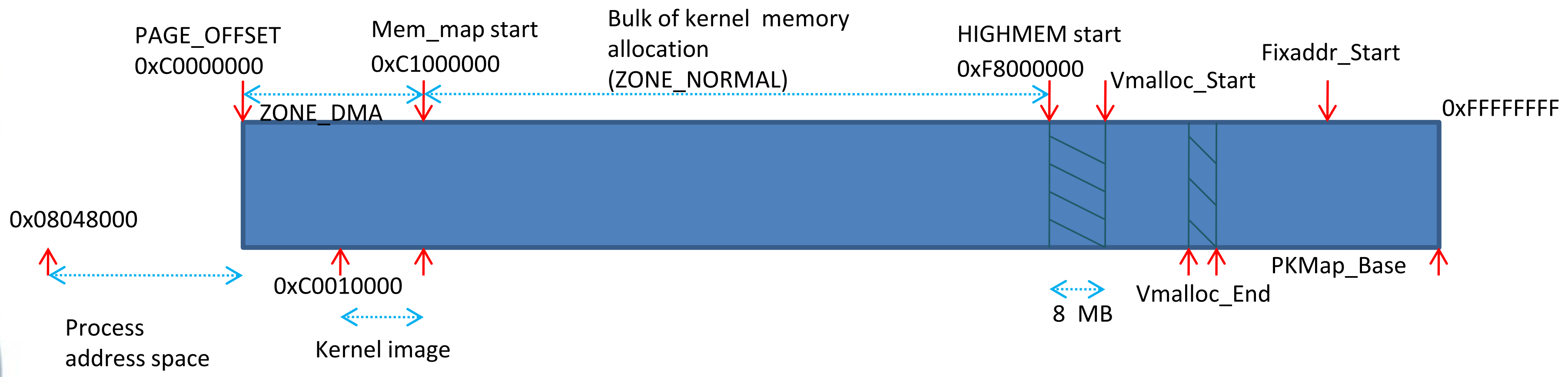
Processes, Threads and Scheduling

Slide Plan

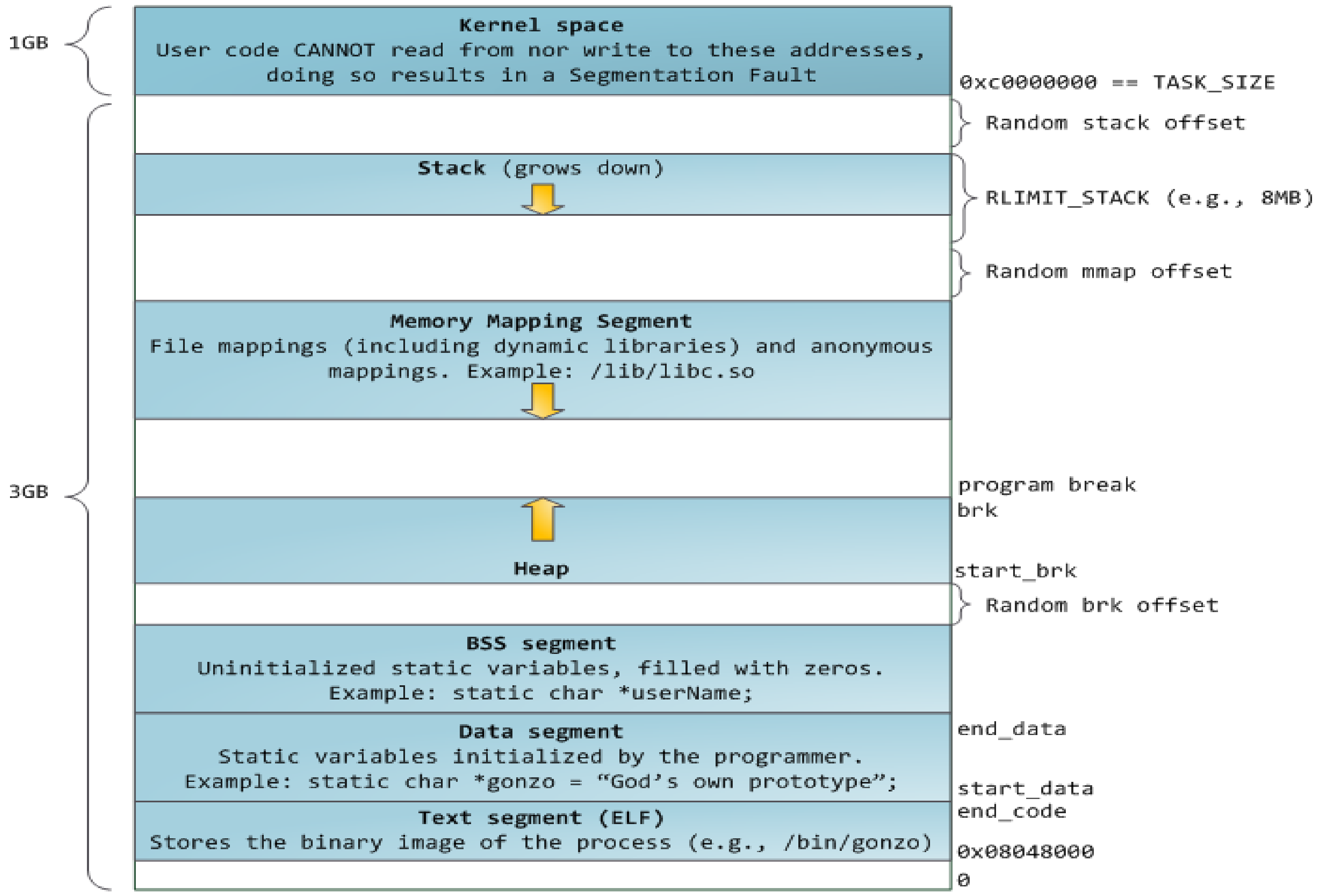
Linux Kernel Virtual Address Space



Note 1: The above addresses are configurable – the HIGHMEM start corresponds to a minimum configurable value for high memory and ZONE_NORMAL extending up to 0xF8000000 corresponds to a maximum value.

Note 2: The kernel image can be configured to start elsewhere. Another typical value for instance places it at the start of ZONE_NORMAL, at 0xC1000000

Process Virtual Address Space (flexible layout)


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

What are the Advantages of the Flexible Layout?

Primarily the flexible layout enables:

- Mmap grows upward from 0x40000000 in the classic layout, and down from RLIMIT_STACK + some random offset in the flexible layout
- The new layout is in essence 'self-tuning' the mmap() and malloc() limits: both malloc() and mmap() can grow until all the address space is full. With the old layout, malloc() space was limited to 900MB, mmap() space to ~2GB, as the heap had a hard upper limit of 0x40000000
- Hence, both malloc(), mmap()/shmat() users to utilize the full address space: 3GB on stock x86, 4GB on x86 4:4 or x86-64 running x86 apps.
- The new layout also allows potentially very large continuous mmap()s
- Address space randomization using a random offset is also included in the flexible layout

What is a Process and what is Process Creation?

- First, clarifications: In a lot of embedded systems, the term 'process' means the same as task/thread, and the terms are used interchangeably. We will use the term 'user thread' only to refer to a user entity that directly corresponds to a kernel schedulable entity and not to the much more rarely used user-level thread with a user space scheduler – an increasingly out of favour idea.

- In more general purpose systems (systems based on Unix), a process is a container of memory:
  - ✓ Process creation sets up a distinct logical-to-physical address set of mappings (fork-exec in Unix/Linux terms)
  - ✓ A main task/thread is created, a task control block(TCB) is allocated, a start function for the task is assigned, stack and heap markers are set up, and the thread is put on a ready queue, i.e. a pointer to the TCB is enqueued on a ready queue

How does a Thread Start Execution?

- Once the current instruction pointer or equivalent of the TCB of the task is set to point to the entry function (generally the main() function of a module), everything is in readiness for execution of application code

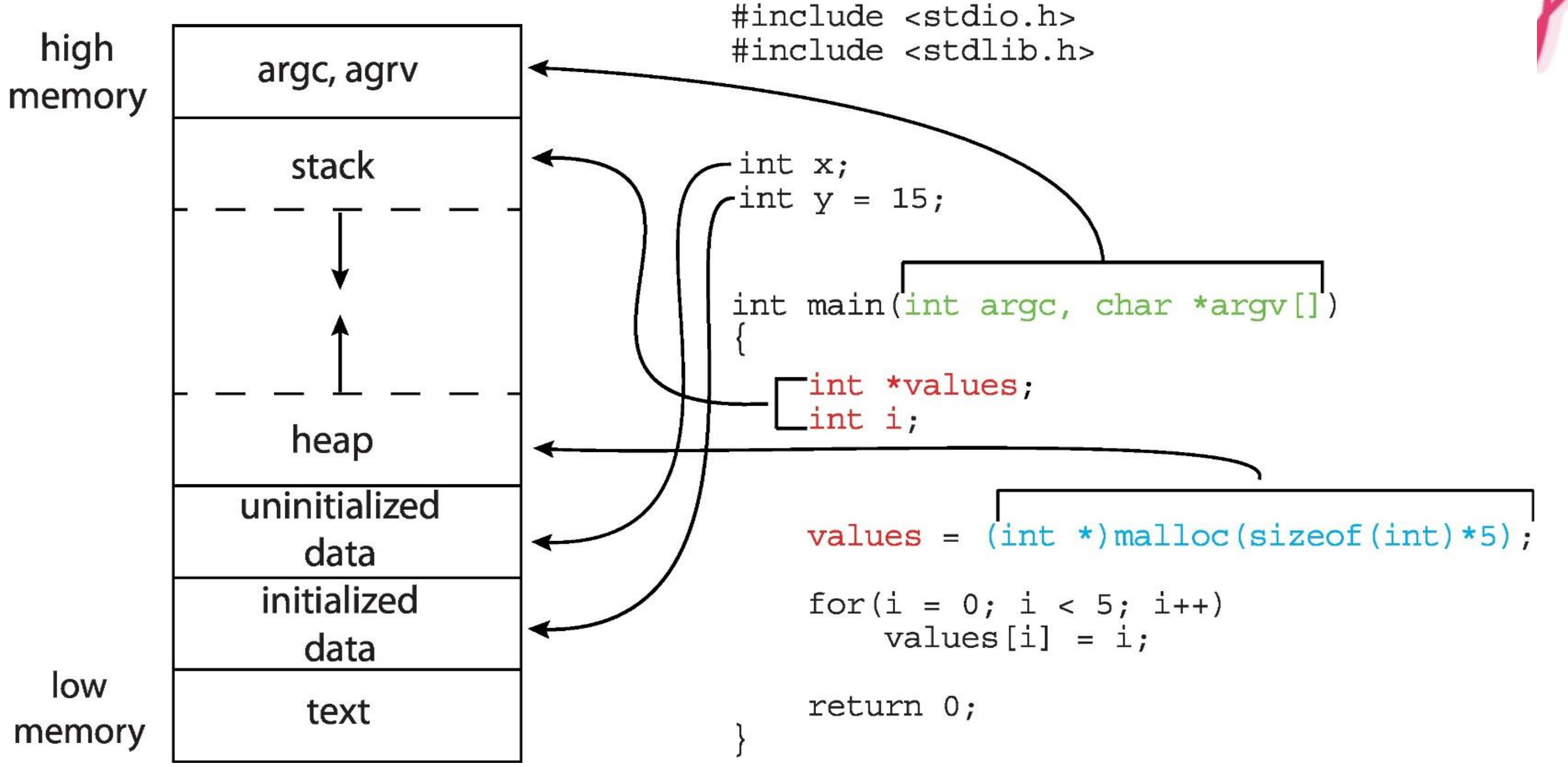
- When the scheduler (based on its algorithm) picks up/dequeues the pointer to the TCB from the ready queue, it also uses a subtle exception-level mechanism of the processor to set things rolling, i.e. make the program counter/instruction register of the processor to point to the entry function of the application on returning from exception – remember, the scheduler code always executes in kernel mode

How does a Thread Start Execution? (Contd.)

- We will look at different scheduler algorithms shortly, but first, let’s take a look at a neat little trick that the processor architecture supports to get the program counter to point to the start function of the application thread

- There are two processor registers that are used as program counter and processor status register backups during exception/interrupt processing. If these are set up as part of the scheduler thread selection function, the return from exception/interrupt instruction will push these backup register values atomically into the program counter and processor status registers.

Memory Layout of a Process – Sample Program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

      values = (int *)malloc(sizeof(int)*5);

      for(i = 0; i < 5; i++)
         values[i] = i;

      return 0;
}
```

What is Process Context?

- There are several kernel data structures that manage the process address space for all threads that run from within the same process. We won’t look at this is in detail for now.

- Let’s look at a list of some items shared across all threads of a process (these make up the process context):
  - ✓ Text segment (instructions)
  - ✓ Data segment (static and global data)
  - ✓ BSS segment (uninitialized data)
  - ✓ Open file descriptors
  - ✓ Signals
  - ✓ Current working directory
  - ✓ User and group IDs

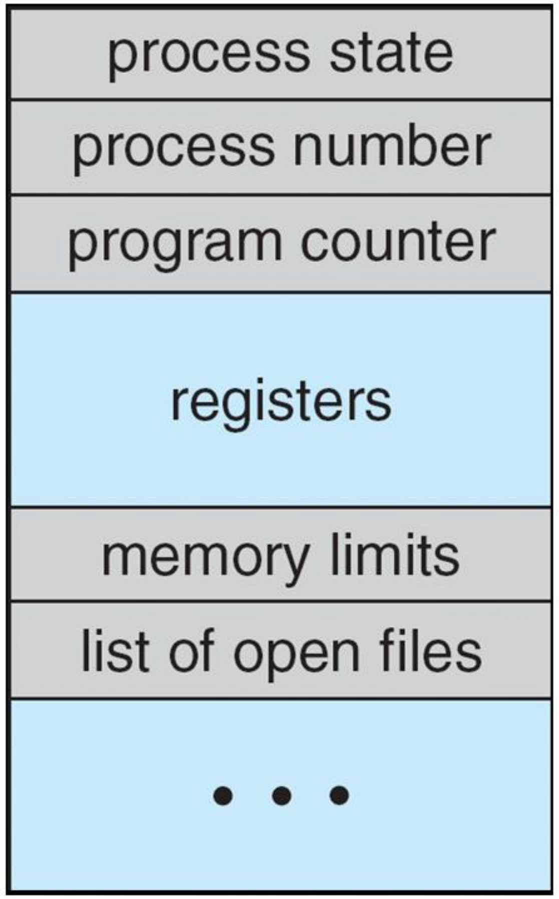
What data is private to a thread? (aka what is thread context?)

- **Threads do not share:**
  - ✓ Stack (local variables on the stack, return addresses, thread local storage)
  - ✓ Processor registers – these need to be saved and restored on thread context switch
  - ✓ Signal masks
  - ✓ Priority (and everything else that gets pushed into a TCB)

Process/Task Control Block - contents

Information associated with each process (also called task control block)

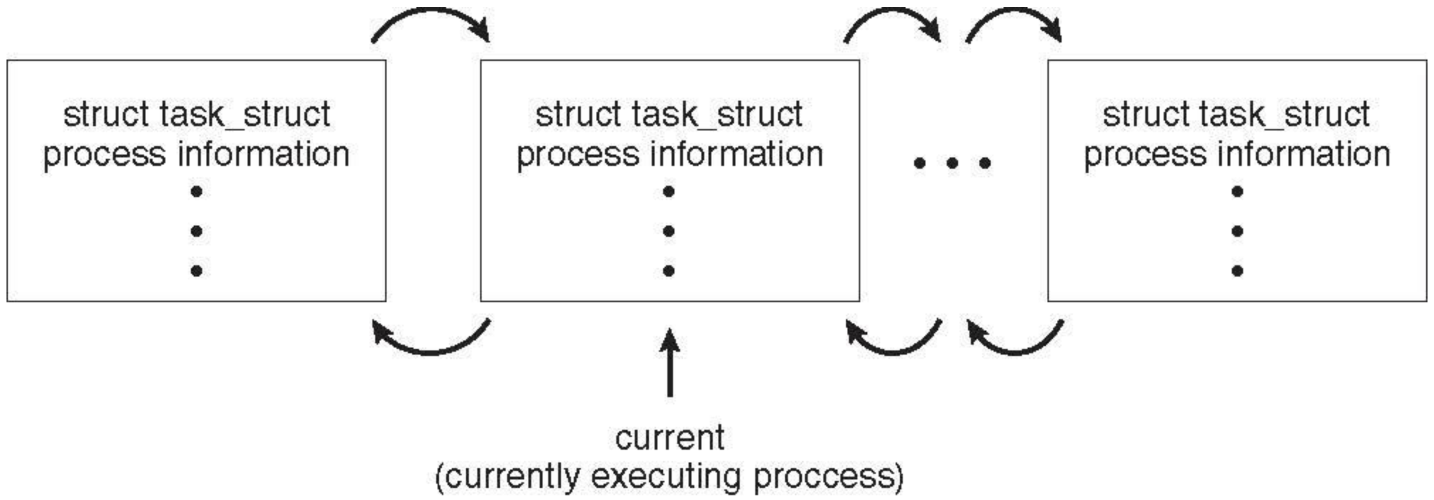
- Process state – Running, waiting, etc.
- Program counter – Location of instruction to execute next
- CPU registers – Contents of all general-purpose and some special-purpose registers
- CPU scheduling information- Priorities, scheduling queue pointers
- Memory-management information – Pointers to memory management data structures of memory allocated to the process
- Accounting information – Time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



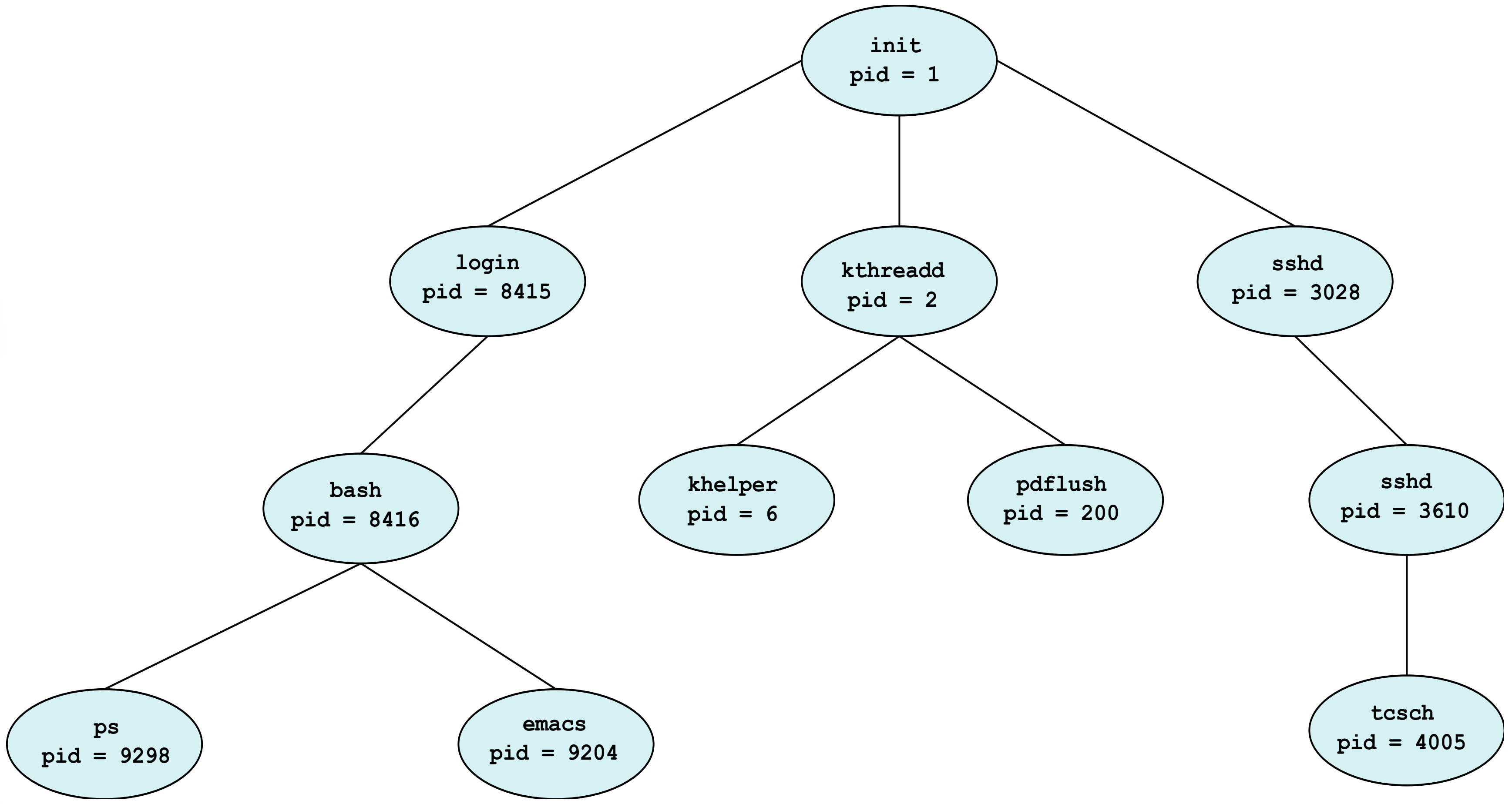
Process Representation (Linux)

Represented by the C structure task_struct

```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space of this
process */
```



A Parent-Child Hierarchy of Processes (Linux)

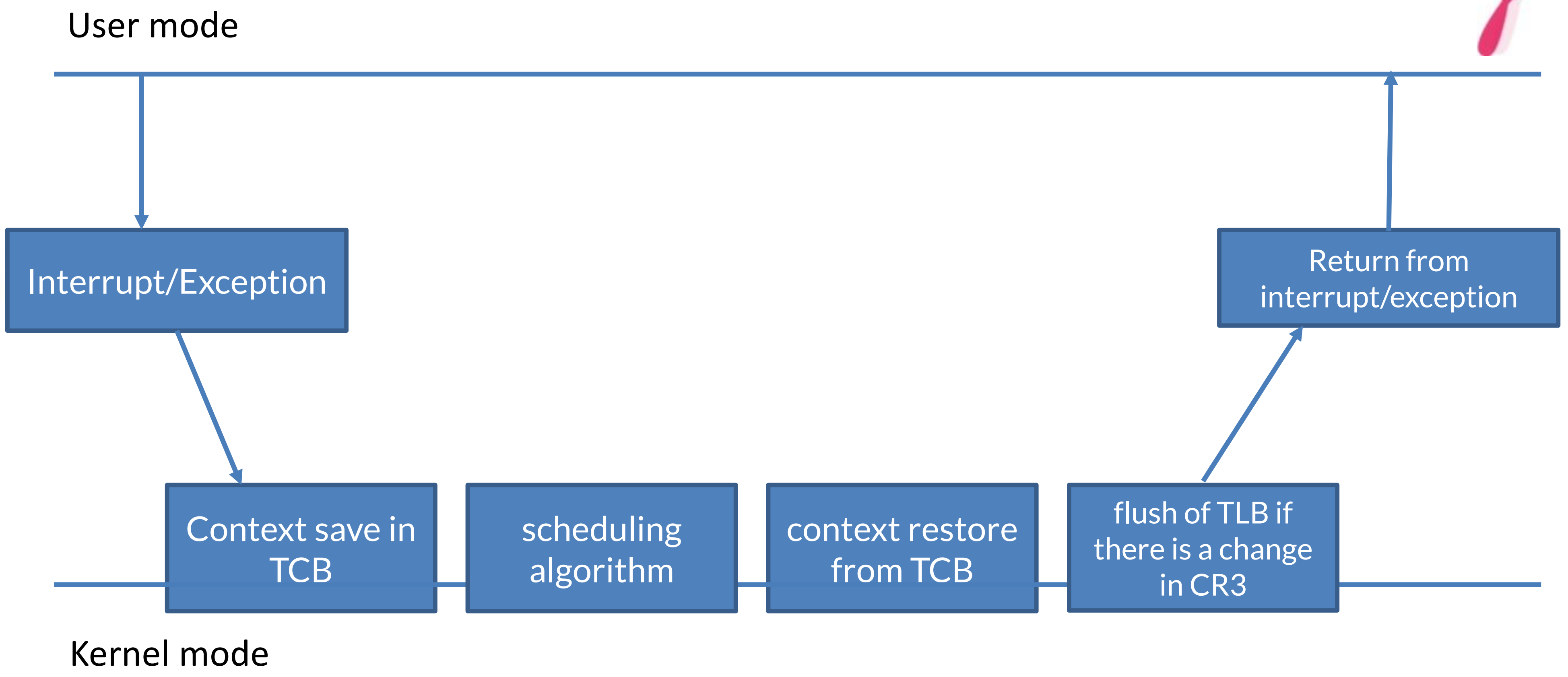


When/how does a thread get scheduled out?

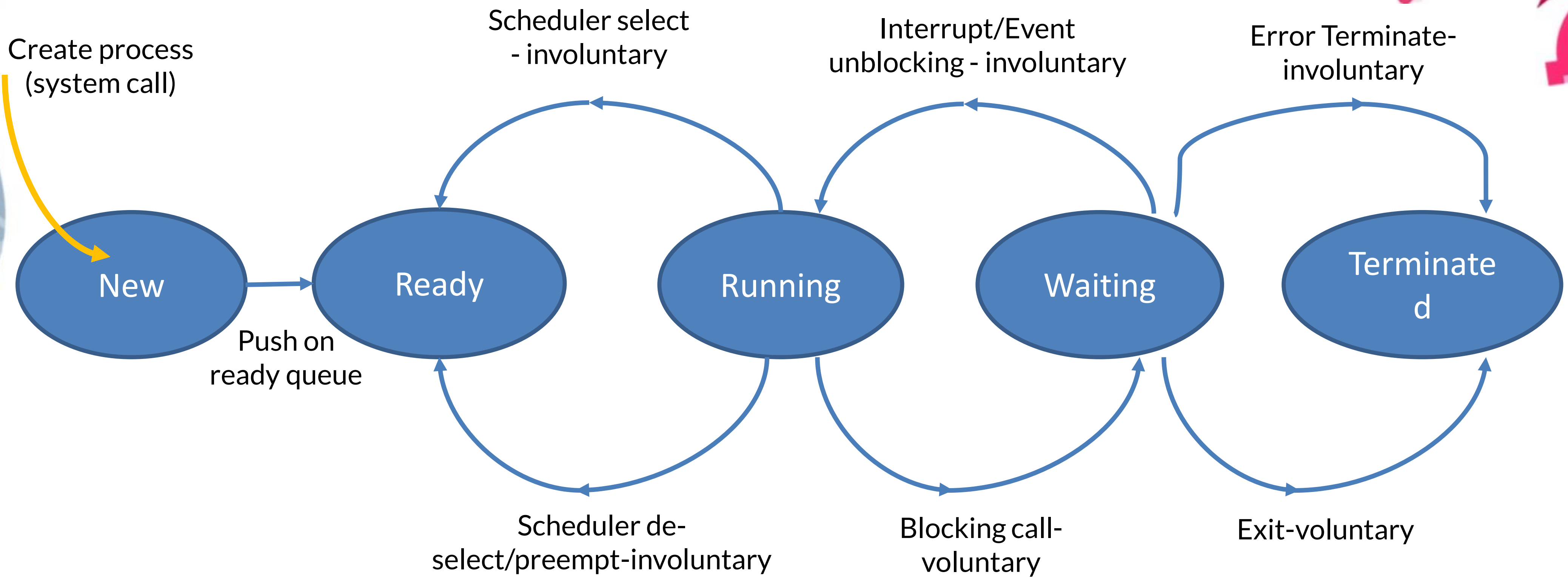
- Voluntarily: By virtue of a blocking system call

- Involuntarily:
  - ✓ An asynchronous interrupt waking up another thread that has execution precedence over the currently running thread according to the scheduling algorithm
  - ✓ A periodic timer interrupt causing the timer interrupt handler to run, that results in threads with more precedence over the currently running thread, being woken up
  - ✓ Or the timer interrupt resulting in the currently running thread to timeslice out on certain OSes

What happens inside the Scheduler?

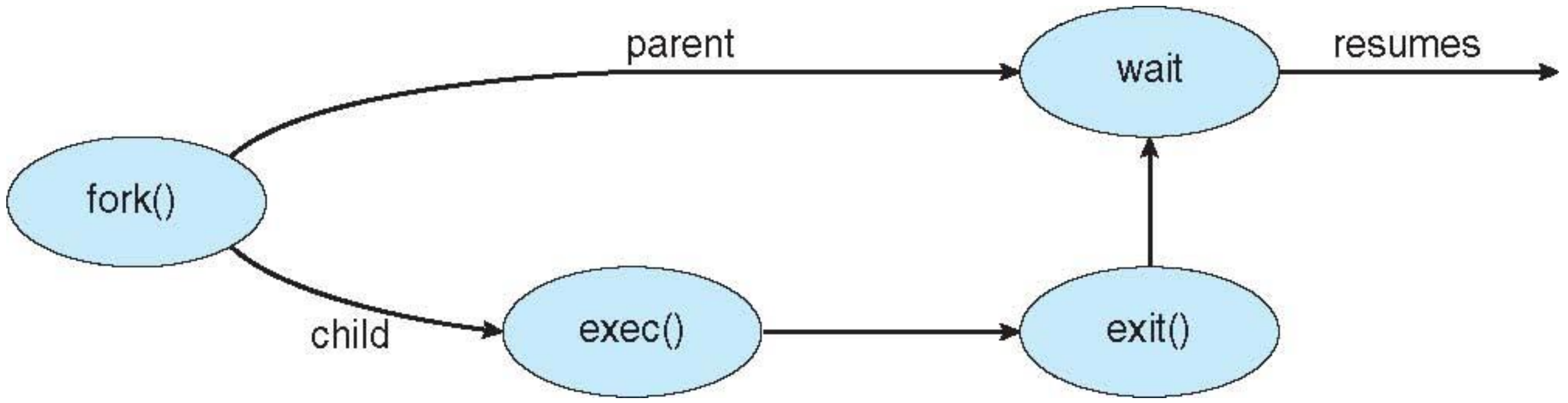


What are the various thread states and how do state transitions happen? – Operation on threads


Create process (system call)
Scheduler select - involuntary
Interrupt/Event unblocking - involuntary
Error Terminate- involuntary
New
Ready
Running
Waiting
Terminated
Push on ready queue
Scheduler de-select/preempt-involuntary
Blocking call- voluntary
Exit-voluntary

Operations on processes/threads (contd.)

- Process/Thread creation: We have dealt with aspects of process/thread (fork-exec) creation as a summary view (slides 6-8). Here is a state diagram that describes the execution sequence of a fork-exec set of routines:



Operations on processes/threads (contd.)

- Process/Thread creation: Let us look at fork-exec in more detail.
- Just to re-iterate: From a scheduler point of view, there is no distinction between a process and a thread entity. When a process is scheduled, it is really its main thread that is being scheduled.
- The difference between a process and a thread is when it comes to the memory and other memory resources - file, device and memory management structures are not shared across processes, while they are, across threads of the same process.

Operations on processes/threads (fork-exec)

- This distinction is particularly sharp when it comes to page tables.
- When a process does a fork, a copy of the page tables is done so that the same logical-to-physical mappings (as the parent process) hold for the newly forked process. In addition, the page table entries (PTEs) of the pages are marked copy-on-write.
- If the fork is followed by an exec of a new binary, then a different set of mappings are created for both code and data.
- If the fork is not followed by an exec, copies of pages are made when they are written to.

Operations on threads (contd.)

- Thread blocking: (Slide 12)

- Thread unblocking: A process gets unblocked and pushed back on the ready queue when the event it was waiting for occurs – either by way of an interrupt (asynchronous hardware event or a timer interrupt) or by way of other threads waking it up by messaging or semaphore release

Operations on threads (contd.)

- Process termination: A process terminates when its main thread terminates:
  - **either voluntarily –**
    - ✓ explicitly: a temporary thread as it completes the execution of its start function, in this case, its main() function calls exit() explicitly or implicitly: the C compiler inserts an exit() at the end of the main() function.
  - or involuntarily –
    - ✓ On hitting an error condition during instruction execution. In which case, the appropriate signal handler (Sigsegv for instance) is called
    - ✓ User termination (cntrl-C) – the Sigterm handler is called
  - Inside the wait of the parent upon return from exit(), task cleanup of the exiting task is done by calling release_task() which in turn, calls put_task_struct() to free the pages containing the process's kernel stack, and deallocate the task_struct

A sample fork-exec-wait sequence

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

When do you Multithread?

Multithreading fetches you most benefits:

- When you need to do parallel independent computation
- When you need to do IO in parallel with computation
- When you need faster responses to several independent events. Then, you break down one big task into several independent units, so that these events don't need to wait for each other for their processing
- When you have multiple cores

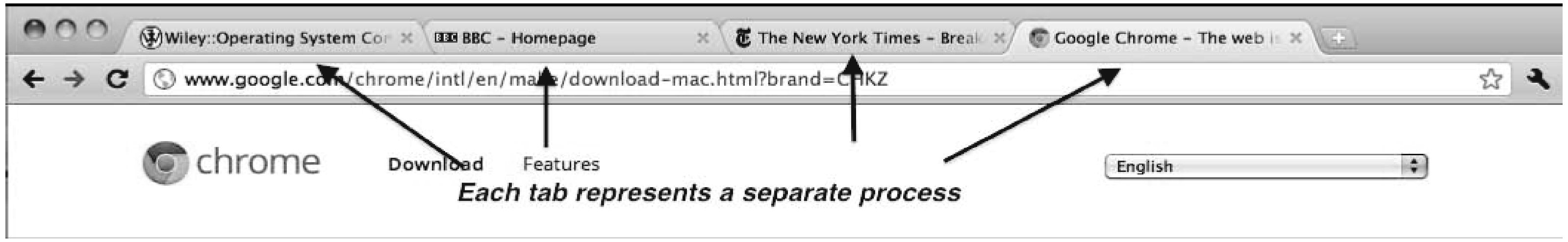
Multi-Process Architecture – Chrome Browser

Many old web browsers ran as a single process (some still do)

If one web site causes trouble, entire browser can hang or crash

The Google Chrome Browser is multi-process with 3 different processes:

- A Browser process that manages the user interface, disk and network I/O
- A Renderer process that renders web pages, deals with HTML, Javascript. A new renderer is created for each website opened
- Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
- A Plug-in process for each type of plug-in


Wiley::Operating System Cor
BBC – Homepage
The New York Times – Break
Google Chrome – The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English
Each tab represents a separate process

What are the different Scheduling Algorithms ?

- There are a whole host of scheduling algorithms in use as part of a variety of OSes. In interests of clarity and brevity, we will look at only three:

  ✓ A fair-share scheduler (Fair-share, time-sharing)

  ✓ A priority-based scheduler (Soft real time)

  ✓ An earliest-deadline-first scheduler (Hard real time)

A Fair-Share Scheduler

Basic ideas:

- A fair-share scheduler gives a certain precedence to higher priority tasks, but doesn't let this totally override considerations of fairness
- Every task starts off with a static priority.
- A fair-share scheduler promotes "good" behaviour, i.e. if a task runs for a relatively short time, and schedules itself out, its "niceness" increases. If it hogs more of the CPU, then its "niceness" decreases.
- This niceness is linked to a dynamic priority, which falls within a short range (say, +/-5) around the static priority

A Fair-Share Scheduler (contd.)

Basic ideas:

- The dynamic priority is the selection criterion that the scheduler uses in choosing the next task to run
- Timeslices are assigned to tasks based on their static priorities
- Timeslices place an upper bound on the continuous time that a task can run (have the CPU for itself)
- So, overall, there is a reasonable amount of fairness leading to lesser scenarios of lower priority tasks getting starved out
- Priorities get importance as well

The Linux O(1) Scheduler

The Linux O(1) scheduler is a good case study of a fair-share scheduler, it ran on several production systems for 5-10 years before being replaced by the current CFS (Completely Fair-Share) scheduler.

Broad scheduling principle:

- Idea: Multi-level (priority) feedback queues and the selection of the highest priority from the priority queue bitmap
- From a source code point of view:
- The schedule() function will swap out the current running thread and swap in the newly chosen thread.

The Linux O(1) Scheduler

- This is done by a one-instruction find of the highest priority bit set in the scheduler bit map array. If time slices have expired meanwhile, the scheduler_tick will set the stage for the current thread to be swapped out. In any case, the selection of the next thread is done by looking up the bitmap in O(1) time.

- The broad objective of schedule() – there are corner cases which we will ignore for now – is to swap the current process out (either voluntarily or because scheduler_tick has explicitly asked for it to be swapped out of the run queue).

The Linux O(1) Scheduler (contd.)

- Static priorities (0-139) (0-99 RT)
- Nice values (dynamic priorities: -20 - +19) (Direct mapping from 0-139 to -20 - +19)
- Time-slices based on static priorities
- Dynamic priorities change -5 - + 5 as a result of “good” behaviour: dynamic priority = max (100, min(static priority – bonus +5, 139), bonus = 0 to 10 depending on average sleep time of process
- Note: Higher priority threads can become interactive faster (with lesser average sleep time than lower priority threads)
- Favour interactivity - interactivity is favoured over time within certain limits imposed by priority
- Selection is based on dynamic priority – the queues are indexed by dynamic priority

The Linux O(1) Scheduler (contd.)

- Two arrays: Active, expired arrays
- Upon expiry of timeslice of individual task, switch it to the expired array, thus working toward lesser starvation of lower-priority tasks
- Favouring interactivity again – if interactive task, then push it back on active array.
- Unless: An older expired process has waited for a long time on the expired array OR If there are higher priority expired processes

The Linux O(1) Scheduler (contd.)

- Real-time threads are always favoured, and they don’t have timeslice limits imposed on them if they are SCHED_FIFO. SCHED_RR does RR among RT threads of same priority.

- Kernel threads are picked for scheduling just like other user threads based on priority – only difference is their mm field points to NULL.

A Priority-Based Scheduler

- A purely priority-based scheduler (VxWorks) is designed to schedule in tasks of higher priority whenever these are ready to run.
- When you have more than one task of the same priority ready to run, then a round-robin policy is used.
- Its design can leave the door open for starvation of lower priority tasks.
- It needs to implement priority-inversion-safe semaphores

Priority Inversion

- Priority inversion happens when you have 3 tasks T1, T2, T3 in a ready to run state, where the priority of T1 > priority of T2 > priority of T3.

- Suppose T3 is running and has taken a resource (a semaphore) that T1 requires and is blocked on, and suppose T2 has pre-empted T3 and doesn't block for a fair while, you have a scenario where a lower priority task T3 has effectively blocked out a higher priority task T1 from running.

- Operating systems that use a priority-based scheduler need to guard against this priority inversion, and boost the priority of lower priority tasks that hold a sema to the priority of the highest priority task that has blocked on the sema

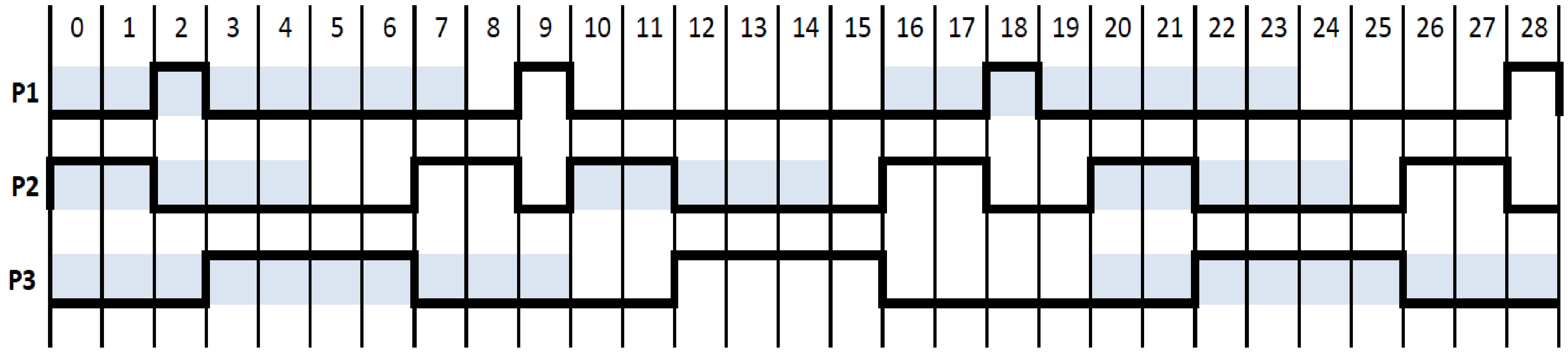
An Earliest-Deadline-First Scheduler

- In this hard-real time scheduler (RTEMS), the highest priority is assigned to the task with the earliest deadline, always.
- This is most suited to systems with hard real-time requirements and where tasks are strictly periodic:
  - ✓ The task whose period is the shortest is assumed to have the earliest deadline
- For instance, if you have 3 tasks P1, P2 and P3 with the following periods and execution times, you will get the following execution pattern with an EDF scheduler

An Earliest-Deadline-First Scheduler

Process Timing Data

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10



Multiprocessor Scheduling

- **When you port a uni-processor scheduler to a multiprocessor-based system, you need to consider the following features:**
  - ✓ Separate local timer interrupts
  - ✓ Separate run queues – why?
  - ✓ Spin lock contention, cores just spin

- **If separate run queues, what if they are un-balanced?**
  - ✓ Rebalance queues periodically(rebalance tick). If idle, pull work off busy queue (idle balance)
  - ✓ You could have some threads assigned core affinity to exploit locality of caching

- **Scheduling domains** – hierarchy of processors for idle balance, rebalance tick:
  - ✓ Local logical core first, same NUMA node next, then only consider different NUMA node

Multiprocessor Scheduling - Porting the O(1) scheduler to an SMP platform

The three streams of execution:

- Voluntary: Call to schedule() – direct invocation – if idle, call *idle_balance* to pull tasks from other run queues. If not idle, go the uniprocessor route of finding the next task from priority bitmap.

- Involuntary: Periodic: Local timer triggered call of scheduler_tick() – lazy invocation, MP point of view -- rebalance_tick periodically rebalances all queues starting with the local SMT thread, pulling threads to the currently executing CPU's queue if required. It is the responsibility of the less busy CPU's to pull tasks to it, the logic being that the busy CPU is already busy. If load_balance fails, kick kernel migration thread to do the job.

- Involuntary: Sporadic: Wake up: *try_to_wake_up*() – lazy invocation. Choice of waking up the thread and putting on either the local core or its previous core (depending on which is busier). If it's a push to the old core, then an IPI is sent to the old core if the newly woken up thread is of higher priority than the currently running thread there.

Interprocess Communication (IPC)

- Now that we have a fair idea of threads and their interactions with each other and other parts of the system (hardware for instance), let’s look at some standard IPC mechanisms:
- ✓ Shared memory
- ✓ Message passing
- ✓ Signaling

Shared Memory

- Shared memory across processes
- Shared memory identifier (shmget – initializes a kernel data structure associated with the particular shared memory id)
- Shared memory attach – shmat – attaches/maps an area in the linear address space of the current process where the shared memory area will map in to
- This could be different addresses in different processes
- The area is contiguous (from a linear/virtual address point of view) – how does this help?
  - ✓ Encourages locality and decreases chances of cache misses as virtually contiguous areas (up to certain size ranges) map to different cache sets
- If the noreserve option is used, when does the actual physical address space get allocated? On demand, i.e. upon a page fault. But, if not enough swap space is available, you could have failures on writes (like overcommit-ed malloc).

Shared Memory (contd.)

- The physical space corresponding to a greater than page size (say, 4k) worth of shared memory need not be physically contiguous – just like malloc.
- External fragmentation -- if you need contiguous logical memory, you will end up having external fragmentation problems – fixed to a certain degree by coalescing free blocks
- What's good:
  - ✓ Ease of programming, flat-address model, once you are done with the creation and attach
  - ✓ More control of data structures you want to use – you could use lock-free data structures for instance
  - ✓ Better best-case performance – if you had a situation where there are more frequent readers than writers (you can choose better performance synchronization methods), or if you could use lock-free data structures
  - ✓ Even in the worst-case with frequent synchronization accesses, there is no copying between kernel and user spaces
- What's bad:
  - ✓ You need to have your own synchronization methods, scalability is a potential issue
- ipcs command - to check shared memory areas (and other IPC profiles)

Shared Memory (contd.)

```
/*Shared memory server*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSize 27
main() {
            char c; int shmid;
            key_t key; char *shm, *s; /* * We'll name our shared memory segment * "5678". */
            key = 5678; /* * Create the segment. */
            if ((shmid = shmget(key, SHMSize, IPC_CREAT | 0666)) <
                        0) { perror("shmget");
                                    exit(1); }
    /* Now we attach the segment to our process address space. */
            if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
                        { perror("shmat");
                                    exit(1); }
    /*  Now put some things into the memory for the other process to read. */
            s = shm; for (c = 'a'; c <= 'z'; c++) *s++ = c; *s = NULL;
        while (*shm != '*')
          sleep(1);
        exit(0);
      }
```

Shared Memory (contd.)

```
/*Shared memory client*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
main() {
                    int shmid;
                    key_t key;
                    char *shm, *s;
/* We need to get the segment named * "5678", created by the server. */
                    key = 5678;
/*  Locate the segment. */
                    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {perror("shmget");
                              exit(1); }
/* Now we attach the segment to our process address space. */
                    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                              perror("shmat");
                              exit(1); }
/* Now read what the server put in the memory. */
                    for (s = shm; *s != NULL; s++) {putchar(*s);
                         putchar('\n');}
      *shm = '*';
      exit(0);
                    }
```

Message Passing

- This is the obvious alternative to Shared Memory
- As the name implies, this is based on messaging using a kernel message queue data structure
- Synchronization and serialized access to the queue is implicit within the send and receive primitives
- Therefore, the designer is spared the need for explicit synchronization
- This is a slightly more complicated programming model than shared memory, but is more scalable

Message Passing (contd.)

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
// structure for message queue
   struct mesg_buffer {
   long mesg_type;
   char mesg_text[100];
} message;
int main()
{
  key_t key;
  int msgid;
   // ftok to generate unique key
   key = ftok("progfile", 65);
```

Message Passing (contd.)

```
// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;
printf("Write Data : ");
gets(message.mesg_text);
// msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);

// display the message
 printf("Data send is : %s \n", message.mesg_text);

  return 0;
}
```

Message Passing (contd.)

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
   long mesg_typet[100];
} message;

int main()
{
   key_t key;
   int msgid;
  // ftok to generate unique key
   key = ftok("progfile", 65);
```

Message Passing (contd.)

```
    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
            message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Signaling

- What is a signal?

- Generating a signal, receiving and handling it
  - ✓ When exactly do you receive a signal?
  - ✓ Default behaviour
  - ✓ Sigterm, Sigsegv
  - ✓ Changing default behaviour
  - ✓ Masking signals
  - ✓ When do you use signals?
    - One at a time, generic task functionality - SIGTERM
    - GDB Simulated OS's scheduler linked to SIGALRM
    - OO event framework building using signals – indexed objects (of different inherited classes) executing their specific member functions based on incoming signal number

Signaling - Internals

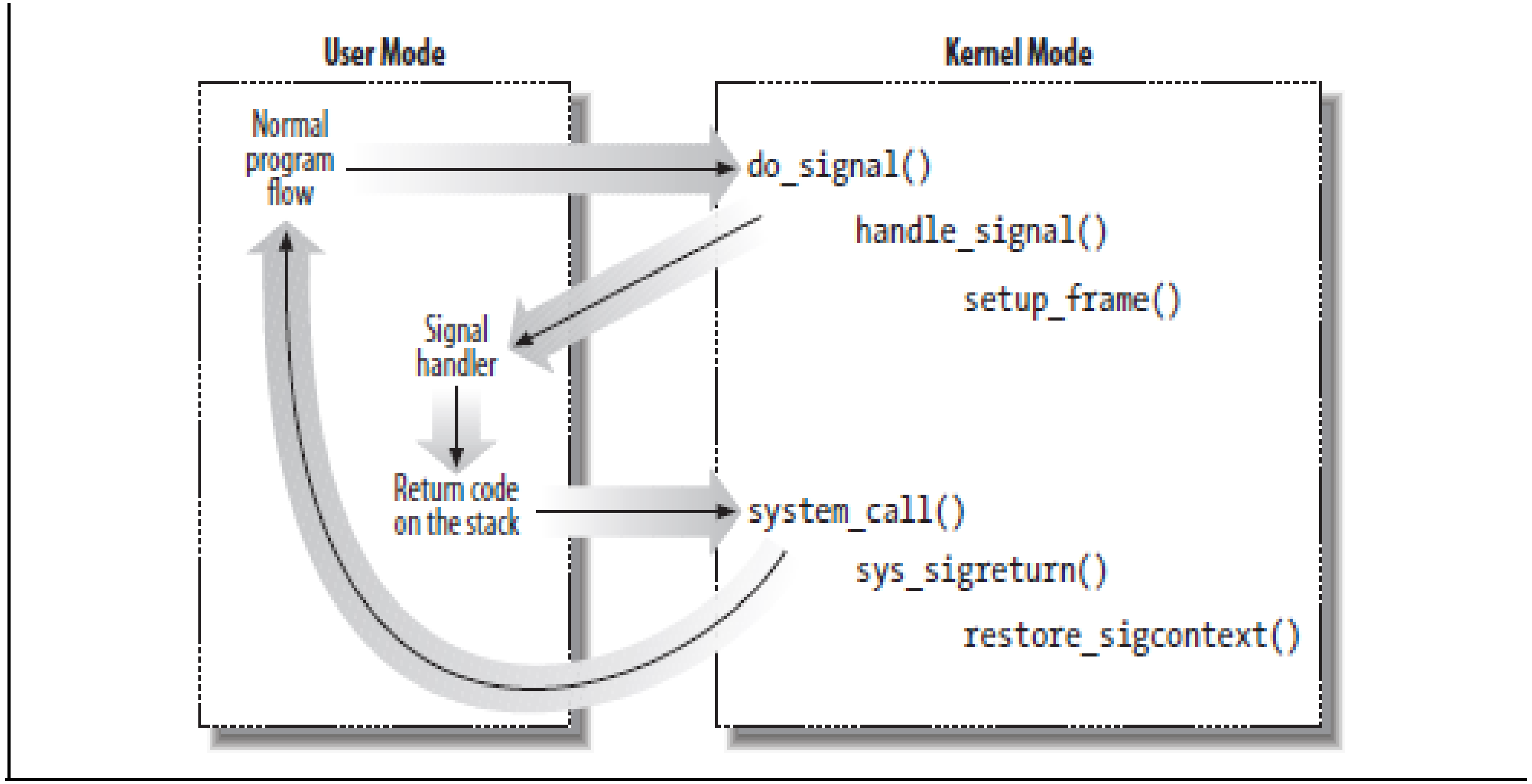
- Playing with the stack at system call resumption/user mode resumption

- Details:
  - ✓ The objective is to execute the signal handler in user mode because that is the way it is expected to execute and the code is part of user space
  - ✓ But, remember the task which is the destination of the signal is either blocked or has been just chosen to run and is either way in kernel mode when the signal has been delivered to it as part of do_signal
  - ✓ The next instruction it will execute is user mode code, next to the one it was executing last
  - ✓ This control flow needs to be changed and the signal handler needs to execute

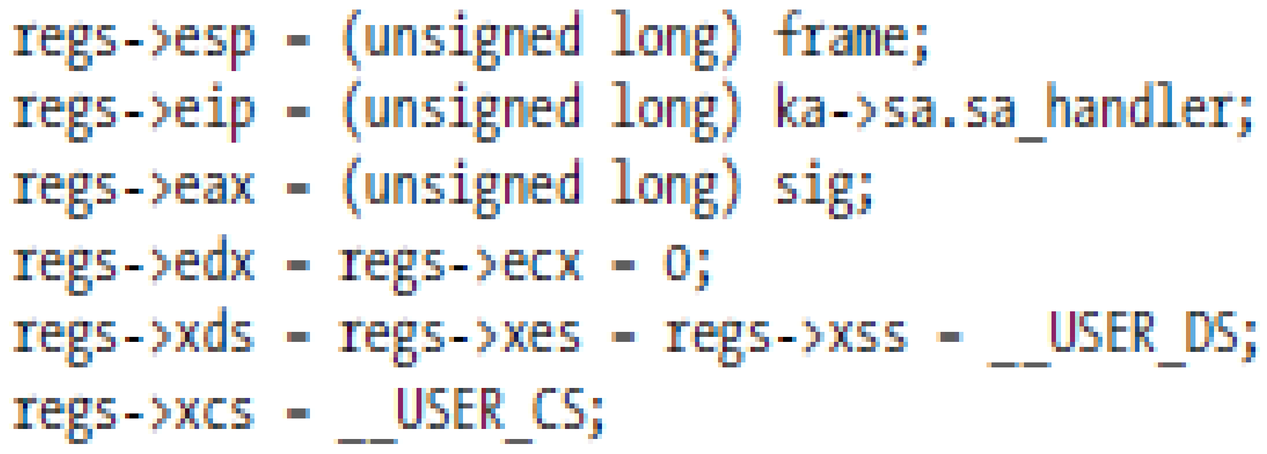
Signaling – Internals (contd.)

- The only way to get this done is to manipulate the kernel mode stack (in setup_frame) where the registers including EIP are saved so that when control goes back to the task, the next instruction that it executes now is the first instruction of the appropriate signal handler.
- The user mode stack is extended by a frame to also include parameters passed to the signal handler and the old user mode context retrieved from the kernel mode stack.
- The return address that needs to be returned to, after executing the signal handler is modified so that control flow takes you to the vsyscall page where you call the sigreturn system call.
- Sigreturn deletes the extended frame after copying the saved hardware context (register area) of the user mode task and restores the kernel mode stack of the user process with this information so that now control flow will return to the user mode application-level code with a return-from-interrupt, after the brief diversion that it took to execute the signal handler.

Signaling – Control Flow

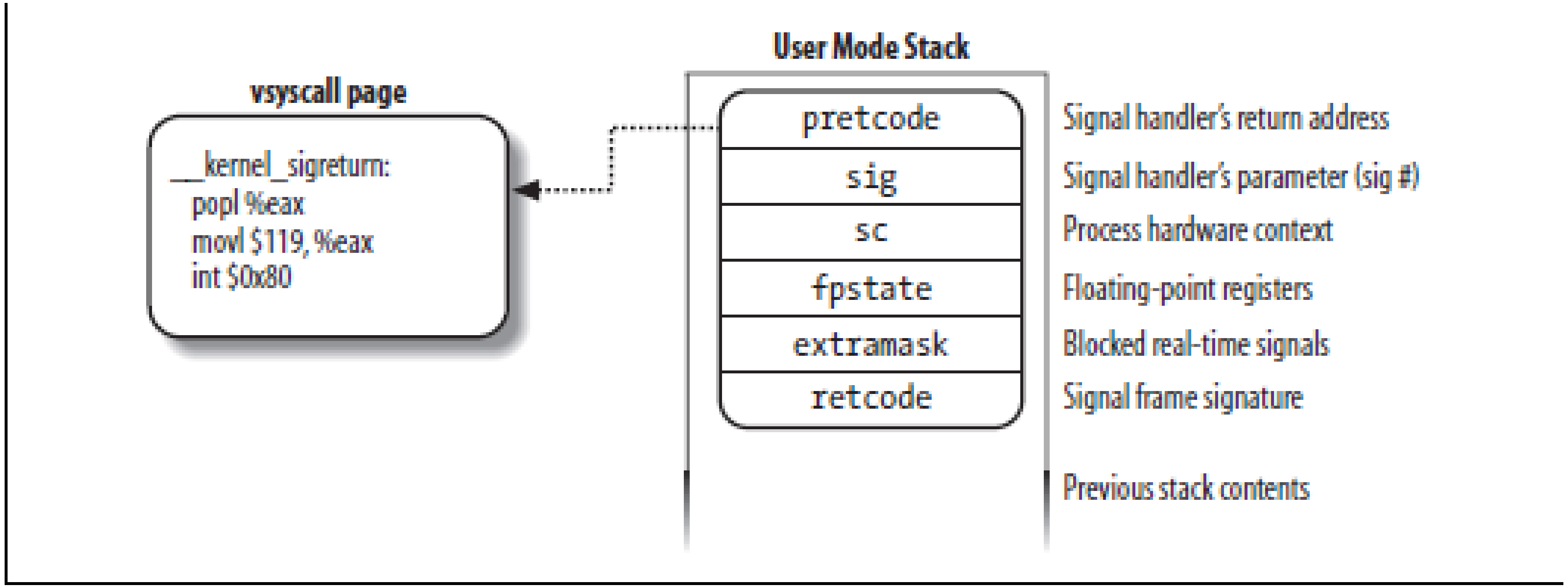


Signaling – Control Flow (contd.)



```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = __USER_DS;
regs->xcs = __USER_CS;
```

Signaling – Using the User mode stack for implementation



Signaling – Overriding SIGINT handling

```
/* A C program that does not terminate when Ctrl+C is pressed */
#include <stdio.h>
#include <signal.h>

/* Signal Handler for SIGINT */
void sigintHandler(int sig_num)
{
  /* Reset handler to catch SIGINT next time.
  signal(SIGINT, sigintHandler);
  printf("\n Cannot be terminated using Ctrl+C \n");
  fflush(stdout);
}
```

Signaling – Overriding SIGINT Handling

```
int main ()
{
   /* Set the SIGINT (Ctrl-C) signal handler to sigintHandler
      Refer http://en.cppreference.com/w/c/program/signal */
   signal(SIGINT, sigintHandler);

   /* Infinite loop */
   while(1)
   {
   }
   return 0;
}
```

Concurrency Models

- Data and task parallelism
- Fork-join parallelism (higher level)
- Mutex, condition variables
- Pthread create example with fork, join, mutex, condition variables

Concurrency Models

A common convention is to look at concurrency from two points of view:

- Data parallelism: Divide a big block of data into independent chunks which can be processed in the same manner in parallel (different sets of rows/columns in matrix multiplication, different parts of the same image etc.)
- Task parallelism: Use different tasks to operate on the same/different chunks of data and assign them to different CPUs or cores (pipelining in image processing).
- The real world picture isn't so clear-cut however, and generally, in most multithreading type scenarios, you have a mix of data and task parallelism.

Fork-and-join

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

Fork-and-join

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Fork-and-join (Joining 10 threads)

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Mutex, Condition Variables Model

- pthread_cond_init(pthread_cond_t *cv);

- pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);
  - ✓ Lock a mutex, change a shared variable, check condition based on variable.
  - ✓ If condition not met, unlock mutex as an implicit part of pthread_cond_wait

- pthread_cond_signal(pthread_cond_t *cv) or pthread_cond_broadcast(pthread_cond_t *cv);
  - ✓ One thread hits condition inside locked mutex region, signals condition variable, then gives up mutex
  - ✓ Thread sleeping in the condition variable wait is woken up and gets the mutex before returning – only one woken-up thread will get the mutex
  - ✓ Re-check the condition while holding the mutex because theoretically, any of the woken-up threads could have changed the condition after getting the mutex
  - ✓ Condition-waiter thread upon successful re-check goes on to unlock the mutex explicitly

Mutex, Condition Variables Model

```
#define SYNC_MAX_COUNT 10
void SynchronizationPoint() {
                    pthread_cond_init(pthread_cond_t *cv); /*init of cond var*/
                    static int sync_count = 0;          /* lock the access to the count */
                    pthread_mutex_lock(&sync_lock); /* increment the counter */          sync_count++;
/* check if we should wait or not */
                    while (sync_count < SYNC_MAX_COUNT) /* wait for the others */
                              pthread_cond_wait(&sync_cond, &sync_lock);
                    else                      /* broadcast that everybody reached the point */
                              pthread_cond_broadcast(&sync_cond);
    /* unlocks the mutex - otherwise only one thread will be able to return from the routine! */
                    pthread_mutex_unlock(&sync_lock); }
```

Thank You!