Control Systems Optimization

Igor Wojnicki

AGH – Univeristy of Science and Technology

2010

Outline

1. Intro
2. Program, Process, Thread

Syllabus

Control Systems Optimization by Introducing Concurrency

- Program, Process, Thread
- IPC: shared memory, semaphores, messages
- Multiple pocessess - why?
- Computer/controller architecture
- Ada language constructs
- Ada IPC
- Functional programming in Erlang
- Erlang: multiprocessing capabilities
- Erlang: IPC
- Control applications examples

Books

- C:
  - Stevens, W. Richard: Advanced Programming in the UNIX Environment, Addison Wesley 2003.
  - Robbins, K.A., Robbins, S.: Practical UNIX Programming, Prentice Hall
- Ada:
  - Barnes, J.: Programming in Ada 2005, Addison Wesley 2006
  - Burns, A.: Concurrent and Real-Time Programming in Ada 2005, Cambridge University Press 2007
- Erlang
  - Armstrong, J.: Programming Erlang: Software for a Concurrent World
  - Cesarini F., Thompson S.: Erlang Programming, O'Reilly 2009

Grading Policy

- Lab Grade
  - Assignments (3)
  - Attendance
  - Do we have an exam?

Programming control systems

- Bare Metal
- OS
- RT OS

Process

- Also called a task
- Execution of an individual program
- Can be traced
  - list the sequence of instructions that execute
- Usually assigned a PID: Process IDentifier

Program, Process, Thread

- Program $\rightarrow$ Process (Thread)
- Process isolation (not always)
- Multiple threads within a process

Execution

- Interleaved
- OS/Scheduler decides which process to run next
- Preemptive vs. Cooperative *Multitasking*
- Process Priorities
- User Process Creation
- Inter-process Communication (IPC)

Process Creation

Principal events that cause process creation

1. System initialization
2. Execution of a process creation system
3. User request to create a new process
4. Initiation of a batch job

Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary): seg fault
4. Killed by another process (involuntary): `kill`

Motivation to use C language

- why use C:
  - simple, low level
  - embedded controlers
  - RT OS
  - different API similar concepts

Processes, C Language

Process handling by system/library calls.

- `fork` – forking a process
- status – process completion
- `wait` – waiting for a process to complete
- shared memory – communication means

fork

```
pid_t fork(void);
```

- Creates a new process by duplicating the calling process.
- The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.
- Return value:
  - PID of the child process is returned in the parent,
  - 0 is returned in the child.
  - On failure, -1 is returned in the parent, no child process is created.

wait

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- wait for state changes in a child of the calling process,
- A state change is considered to be:
  - child terminated; the child was stopped by a signal;
  - or the child was resumed by a signal.
- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; beware of a *zombie* child.
- If a child has already changed state, then these calls return immediately.
- Otherwise they block.

Shared Memory

- Can be accessed by multiple processes
- Permissions can be defined
- system calls: `shmget()`, `shmat()`, `shmdt()`, `shmctl()`

shmget()


```
int shmget(key_t key, size_t size, int shmflg);
```

- Returns the identifier of the shared memory segment associated with the value of the argument key.

shmat(), shmdt()

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- Attaches the shared memory segment identified by shmid to the address space of the calling process.

```
int shmdt(const void *shmaddr);
```

- Detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process.

shmctl()

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Performs the control operation specified by cmd on the shared memory segment whose identifier is given in shmid.

Shared Memory and Process Handling Example I

```
int main(void)
{
  int  shmid, *shmptr;
  pid_t pid;
  int status;
  int i;

  /* request shared mem */
  shmid=shmget(IPC_PRIVATE,sizeof(int),SHM_R | SHM_W);
  printf("Shared memory id: %d\n",shmid);
  /* attach a ptr to shm */
  shmptr=(int *)shmat(shmid,0,0);
  *shmptr=0;
  printf("start: %d\n",*shmptr);
```

Shared Memory and Process Handling Example II

```
/* detach a ptr from shm */
shmdt(shmptr);

for (i=0; i<5; i++) {
  pid=fork();
  if (pid==0){
    /* child code */
    shmptr=(int *)shmat(shmid,0,0);
    printf("child: %d\n",*shmptr);
    (*shmptr)++;
    shmdt(shmptr);
    return(0);
  }
}
```

Shared Memory and Process Handling Example III

```
  while (wait(&status)!=-1);

  shmptr=(int *)shmat(shmid,0,0);
  printf("stop: %d\n",*shmptr);
  shmdt(shmptr);
  /* remove shared mem */
  shmctl(shmid,IPC_RMID,0);
  return 0;
}
```